Towards Neural-Guided Program Synthesis for Linear Temporal Logic Specifications

Alberto Camacho
University of Toronto
Vector Institute
acamacho@cs.toronto.edu

Sheila A. McIlraith
University of Toronto
Vector Institute
sheila@cs.toronto.edu

Abstract

Synthesizing a program that realizes a logical specification is a classical problem in computer science. We examine a particular type of program synthesis, where the objective is to synthesize an agent strategy that reacts to a potentially adversarial environment while ensuring that all executions satisfy a *Linear Temporal Logic* (LTL) specification. Unfortunately, exact methods to solve so-called *LTL synthesis* via logical inference do not scale. In this work, we cast LTL synthesis as an optimization problem. We employ a neural network to learn a Q-function that is then used to guide search, and to construct programs that are subsequently verified for correctness. Our method is unique in combining search with deep learning to realize LTL synthesis. In our experiments the learned Q-function provides effective guidance for synthesis problems with relatively small specifications, while accommodating large specifications that defy formal methods.

1 Introduction

Automated synthesis of programs from logical specification – *program synthesis from specification* – is a classic problem in computer science (Church, 1957). We are concerned with *LTL synthesis* – the synthesis of a reactive module that interacts with the environment such that all executions satisfy a prescribed *Linear Temporal Logic* (LTL) formula. The formula specifies the objective of the program or agent and other sundry constraints including assumptions on the behavior of the environment (Pnueli and Rosner, 1989). Synthesized programs are *correct by construction*. LTL synthesis has a myriad of applications including the automated construction of logical circuits, game agents, and controllers for intelligent devices and safety-critical systems.

Programming from specification, and specifically LTL synthesis has been studied intensely in theory and in practice (e.g., (Jacobs et al., 2019)). We contrast it with *programming by example* which synthesizes a program that captures the behavior of input-output data or program traces (e.g., (Gulwani, 2016)). In this category, differential approaches use deep learning for *program induction*, to *infer* the output of the program for new inputs (e.g. (Ellis et al., 2016; Parisotto et al., 2017; Zhang et al., 2018; Chen et al., 2018)). In neural-guided search, statistical techniques are used to guide search (e.g. (Balog et al., 2017)), motivated in part by the fact that discrete search techniques may be more efficient than differentiable approaches alone (Gaunt et al., 2016).

LTL synthesis is a challenging problem. Its complexity is known to be 2EXP-complete, and exact methods to solve this problem via logical inference do not scale – in part because of the prohibitively large search space. Thus, there is a need for exploration and development of novel approaches to LTL synthesis that have the potential to scale.

In this work we make a first step towards applying two scalable techniques to LTL synthesis. Namely, *search* and *learning*. We are interested in answering this question:

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

Can we learn good guidance for LTL *synthesis?*

Sophisticated search techniques, and in particular heuristic search, have the potential to scale to large spaces by means of efficient exploration and pruning. As to date, unfortunately, there are no heuristics specially designed for LTL objectives, and existing heuristics are mostly designed to provide guidance in deterministic environments – not the case of LTL synthesis. Recent successes of deep learning for Markov Decision Processes (e.g. (Mnih et al., 2015)) and complex sequential decision-making problems (e.g. (Silver et al., 2018)) have inspired us to think that neural networks may be able to capture the structure of LTL synthesis specifications, and be used to provide guidance to search methods.

The main challenge that we had to address in this work was how to frame LTL synthesis – a problem that has commonly been addressed via logical inference by formal methods – into an optimization problem. To this end, we introduce a novel dynamic programming based formulation of LTL synthesis that allows us to deploy statistical approaches, and in particular deep learning combined with search, to realize LTL synthesis. We employ a neural network to learn a Q-function that is used to guide search, and to construct programs that are subsequently verified for correctness, thereby benefitting from the scalability of an approximate method while maintaining correct-by-construction guarantees.

In our experiments the learned Q-function provided effective guidance for the synthesis of relatively small specifications, solving a number of the community benchmarks.

2 Reactive LTL Synthesis

The central problem that we examine in this paper is the synthesis of controllers for sequential decision-making in discrete dynamical environments. *Reactive synthesis* constructs a strategy such that *all* executions of the strategy realize a specified temporally extended property, regardless of how the environment behaves (Pnueli and Rosner, 1989). The specified property may include reachability goals and safety and liveness properties, as well as assumptions regarding the behavior of the environment. In contrast with MDPs, the environment dynamics are not stochastic nor necessarily Markovian but rather non-deterministic.

In LTL synthesis, the temporally extended property to be satisfied takes the form of a *Linear Temporal Logic* (LTL) formula over a set of *environment* ($\mathcal{X}$) and *system* ($\mathcal{Y}$) variables (Definition 1). LTL is a modal logic that extends propositional logic with temporal modalities to express temporally extended properties of infinite-length state traces. In a nutshell, $\bigcirc\varphi$ denotes that φ holds in the next timestep, and $\varphi_1\mathcal{U}\varphi_2$ denotes that φ_1 holds until φ_2 holds. Unary operators *eventually* ($\Diamond$) and *always* ($\Box$), and binary operator *release* ($\mathcal{R}$) can be defined using $\bigcirc$ and $\mathcal{U}$ (cf. (Pnueli, 1977)). Formally, we say that an infinite trace $\rho = s_1, s_2, \ldots$ satisfies φ ($\rho \models \varphi$, for short) iff $\rho, 1 \models \varphi$, where for every natural number $i \geq 1$:

- $\rho, i \models p$, for a propositional variable p, iff $p \in s_i$,
- $\rho, i \models \neg\psi$ iff it is not the case that $\rho, i \models \psi$,
- $\rho, i \models (\psi \wedge \chi)$ iff $\rho, i \models \psi$ and $\rho, i \models \chi$,
- $\rho, i \models \bigcirc\varphi$ iff $\rho, i+1 \models \varphi$,
- $\rho, i \models \varphi\mathcal{U}\psi$ iff there exists a $j \geq i$ such that $\rho, j \models \psi$, and $\rho, k \models \varphi$ for every $i \leq k < j$.

Definition 1. *An* LTL *specification is a triplet $\langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$, where $\mathcal{X}$ and $\mathcal{Y}$ are two finite disjoint sets of environment and system variables, respectively, and φ is an* LTL *formula over $\mathcal{X} \cup \mathcal{Y}$.*

Definition 2. *The* synthesis *problem for an* LTL *specification $\langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ consists in computing an agent strategy $f : (2^{\mathcal{X}})^+ \to 2^{\mathcal{Y}}$ such that, for any infinite sequence $X_1X_2\cdots$ of subsets of $\mathcal{X}$, the sequence $(X_1 \cup f(X_1))(X_2 \cup f(X_1X_2))\cdots$ satisfies φ. The* realiazability *problem consists in deciding whether one such strategy exists.*

Example: The LTL specification $\langle \{x\}, \{y\}, \Box(x \leftrightarrow \bigcirc y) \rangle$ is realizable. One agent strategy that synthesizes the specification outputs $Y_{n+1} = \{y\}$ whenever $X_n = \{x\}$, and $Y_{n+1} = \emptyset$ whenever $X_n = \emptyset$. Note, the LTL formula does not constrain the output of the system in the first timestep.

2.1 Bounded Synthesis

Traditional approaches to LTL realizability and synthesis rely on automata transformations of the LTL formula. Here, we review results from so-called *bounded synthesis* approaches, that make use of *Universal Co-Büchi Word* (UCW) automata (Kupferman and Vardi, 2005). An important result in bounded synthesis is that LTL realizability can be characterized in terms of the runs of UCW automata, in a way that the space of the search for solutions can be *bounded*. The result, adapted from (Schewe and Finkbeiner, 2007), is stated in Theorem 1. Besides that, the computational advantage of bounded synthesis is that UCW automata transformations can be done more efficiently in practice than to other types of automata (e.g., parity automata).

UCW automata: An UCW automaton is a tuple $\langle Q, \Sigma, q_0, \delta, Q_F \rangle$, where Q is a finite set of states, Σ is the input alphabet (here, $\Sigma := 2^{\mathcal{X} \cup \mathcal{Y}}$), $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to 2^Q$ is the (non-deterministic) transition function, and $Q_F \subseteq Q$ is a set of rejecting states. Without loss of generality, we assume $\delta(q, \sigma) \neq \emptyset$ for each $(q, \sigma) \in Q \times \Sigma$. A *run* of A_φ on a play $\rho = (X_1 \cup Y_1)(X_2 \cup Y_2) \cdots$ is a sequence $q_0 q_1 \cdots$ of elements of Q, where each q_{i+1} is an element of $\delta(q_i, X_i \cup Y_i)$. The *co-Büchi* index of a run is the maximum number of occurrences of rejecting states. A run of A_φ is *accepting* if its co-Büchi index is finite, and a play ρ is *accepting* if all the runs of A_φ on ρ are accepting. An LTL formula φ can be transformed into an UCW automaton that accepts all and only the models of φ in worst-case exponential time (Kupferman and Vardi, 2005).

Theorem 1. *Let $\mathcal{A}_\varphi$ be a UCW transformation of* LTL *formula φ. An* LTL *specification $\langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ is realizable iff there exists a strategy $f : (2^{\mathcal{X}})^+ \to 2^{\mathcal{Y}}$ and $k < \infty$, worst-case exponential in the size of $\mathcal{A}_\varphi$, such that, for any infinite sequence $X_1 X_2 \cdots$ of subsets of $\mathcal{X}$, all runs of $\mathcal{A}_\varphi$ on the sequence $(X_1 \cup f(X_1))(X_2 \cup f(X_1 X_2)) \cdots$ hit a number of rejecting states that is bounded by k.*

2.2 Automata Decompositions

UCW transformations of LTL formulae are worst-case exponential, and can be a computational bottleneck. To mitigate for this, synthesis tools Acacia$^+$ (Bohy et al., 2012) and SynKit (Camacho et al., 2018b) decompose the formula into a conjunction of subformulae, and then transform each subformula into UCW automata. Clearly, each subformula is potentially easier to transform into UCW automata than the whole formula. The results for bounded synthesis can be naturally extended to multiple automata decompositions, where the automata capture, collectively, LTL satisfaction.

Theorem 2 (adapted from (Bohy et al., 2012))**.** *Let $\mathcal{A}_i$ be UCW transformations of* LTL *formulae φ_i, $i = 1..m$, and let $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_m$.* LTL *specification $\langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ is realizable iff there exists a strategy $f : (2^{\mathcal{X}})^+ \to 2^{\mathcal{Y}}$ and $k < \infty$ such that, for any infinite sequence $X_1 X_2 \cdots$ of subsets of $\mathcal{X}$, the runs of each $\mathcal{A}_i$ on the sequence $(X_1 \cup f(X_1))(X_2 \cup f(X_1 X_2)) \cdots$ hit a number of rejecting states that is bounded by k.*

3 Safety Games for Bounded Synthesis

Reactive synthesis is usually interpreted as a two-player game between the *system* player, and the *environment* player. In each turn, the environment player makes a move by selecting a subset of *uncontrollable* variables, $X \subseteq \mathcal{X}$. In response, the system player makes a move by selecting a subset of *controllable* variables, $Y \subseteq \mathcal{Y}$ – thus, the sets of player actions are the powersets of $\mathcal{X}$ and $\mathcal{Y}$. Game states and transitions are constructed by means of *automata* transformations of the LTL formula. In particular, bounded synthesis is interpreted as a *safety game* where the agent is constrained to react in a way that (infinite-length) game plays must yield automaton runs that hit a bounded number of rejecting states (cf. Theorems 1 and 2). We formalize safety games below, and frame bounded synthesis as safety games in Section 3.1.

Safety games: In this paper, a two-player *safety game* is a tuple $\langle Z_{env}, Z_{sys}, S, s_1, T, S_{bad} \rangle$, where Z_{env} and Z_{sys} are sets of actions, S is a set of states, $s_1 \in S$ is the initial state of the game, $T : S \times Z_{env} \times Z_{sys} \to S$ is a transition function, and $S_{bad} \subseteq S$ is a set of losing states. Analogously to LTL synthesis, we refer to the players as the environment and system. The game starts in s_1, and is played an infinite number of turns. In each turn, the environment player selects an action $x \in Z_{env}$, and the system player reacts by selecting an action $y \in Z_{sys}$. If the game state in the n-th turn is s_n and the players moves are x_n and y_n, then the game state transitions to $s_{n+1} = T(s_n, x_n, y_n)$. Thus, game plays are infinite sequences of pairs $(x, y) \in Z_{env} \times Z_{sys}$ that yield sequences of game states $\rho = s_1 s_2 \cdots$. A game play is winning (for the system player) if it yields a sequence of states that never hits a losing state. Solutions to a safety game are *policies* $\pi : S \times Z_{env} \to Z_{sys}$ that guarantee that game plays are winning, regardless how the environment moves, provided that the system acts in each state as mandated by π. Safety games can be solved in polynomial time in the size of the search space, via fix-point computation of the set of *safe states* – i.e., states from which the system player has a winning strategy to avoid falling into losing states. If the initial state is a safe state, then a winning strategy for the system player can be obtained by performing actions that prevent the environment from transitioning into an unsafe state.

3.1 Safety Game Reformulations

Bounded synthesis approaches reduce the synthesis problem into a series of safety games G_k, parametrized by k. In those games, the environment and system players perform actions that correspond to the powersets of $\mathcal{X}$ and $\mathcal{Y}$ variables, respectively. First, φ is transformed into a UCW automaton, $A_\varphi = \langle Q, \Sigma, q_0, \delta, Q_F \rangle$. With a fixed order in the elements of $\mathcal{X}$, $\mathcal{Y}$, and Q, we identify each subset $X \subseteq \mathcal{X}$ with a boolean vector $\boldsymbol{x} = \langle x^{(1)}, \ldots, x^{(|\mathcal{X}|)} \rangle$ that indicates whether $x_i \in X$ by making the ith element in $\boldsymbol{x}$ true (similarly with subsets of $\mathcal{Y}$ and Q). Game states are vectors $\boldsymbol{s} = \langle s^{(1)}, \ldots, s^{(|Q|)} \rangle$ that do bookkeeping of the co-Büchi indexes of the runs of A_φ on the partial play that leads to $\boldsymbol{s}$. More precisely, each element $s^{(i)}$ is an integer that tells the maximum co-Büchi index among all the runs of A_φ on the partial play that finish in $q^{(i)}$. If none of the runs finish in $q^{(i)}$, then $s^{(i)} = -1$. The transition function, T, progresses the automaton runs captured in a state $\boldsymbol{s}$ according to the move of the environment and system players.

$$
\begin{aligned}
T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y}) &\coloneqq \langle s'^{(1)}, \ldots, s'^{(|Q|)} \rangle, \text{ where} \\
t^{(i)} &\coloneqq \max\{ s^{(j)} \mid q_i \in \delta(q_j, \boldsymbol{x} \| \boldsymbol{y}) \} \\
s'^{(i)} &\coloneqq t^{(i)} + \mathbb{1}(q_i \in Q_F) \cdot \mathbb{1}(t^{(i)} > -1)
\end{aligned}
$$

We write $\mathrm{idx}(\boldsymbol{s})$ to refer to the *co-Büchi index of game state* $\boldsymbol{s}$, that we define as the maximum co-Büchi index among all the runs captured by $\boldsymbol{s}$. Formally, $\mathrm{idx}(\boldsymbol{s}) \coloneqq \max_i s^{(i)}$. Losing states in S_{bad} are those $\boldsymbol{s}$ with $\mathrm{idx}(\boldsymbol{s}) = k$. Intuitively, in those games the environment player aims at maximizing the index of the states visited, whereas the system player aims at keeping this number bounded. Theorem 3 relates LTL realizability with the existence of solutions to those safety games. For computational reasons, we redefine a transition function T_k that reduces the search space to those states with co-Büchi index bounded by k.

Theorem 3 (adapted from (Schewe and Finkbeiner, 2007))**.** LTL *specification* $\langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ *is realizable iff the safety game* $G_k = \langle 2^{\mathcal{X}}, 2^{\mathcal{Y}}, S, \boldsymbol{s_1}, T_k, S_{bad} \rangle$ *constructed from any UCW transformation of* φ*,* $A_\varphi = \langle Q, \Sigma, q_0, \delta, Q_F \rangle$*, has a solution for some* $k < \infty$*, worst-case exponential in the number of states in* A_φ*, where:*

$$
\begin{aligned}
S &\coloneqq [-1, k]^Q \\
\boldsymbol{s_1} &\coloneqq \langle 0, -1, \ldots, -1 \rangle \\
T_k(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y}) &\coloneqq \langle s'^{(1)}, \ldots, s'^{(|Q|)} \rangle, \textit{ where} \\
s'^{(i)} &\coloneqq \min(k, T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y})^{(i)}) \\
S_{bad} &\coloneqq \{ \boldsymbol{s} \in S \mid \mathrm{idx}(\boldsymbol{s}) = k \}
\end{aligned}
$$

3.2 Automata Decompositions

Theorem 3 can be extended to handle multiple automata decompositions of the LTL formula. The result, stated in Theorem 4, is adapted from known results that proof the correctness of bounded synthesis approaches employed by Acacia$^+$ and SynKit, which exploit decompositions to improve scalability (Bohy et al., 2012; Camacho et al., 2018b). We presume the LTL specification formula is a conjunction of m subformulae $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_m$. For each $i = 1..m$, let $G_k^{(i)}$ be the safety game constructed as described in Theorem 3 from a UCW automaton transformation $\mathcal{A}_i$ of φ_i. The idea is to construct a cross-product safety game G_k that maintains the dynamics of each $G_k^{(i)}$ in parallel.

Theorem 4. *Let $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_m$ be an* LTL *formula, and let $G_k^{(i)} = \langle 2^{\mathcal{X}}, 2^{\mathcal{Y}}, S^{(i)}, \boldsymbol{s_1^{(i)}}, T_k^{(i)}, S_{bad}^{(i)} \rangle$ be safety games associated to each φ_i, $i = 1..m$, and constructed as described in Theorem 3.* LTL *specification $\langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ is realizable iff the safety game $G_k = \langle 2^{\mathcal{X}}, 2^{\mathcal{Y}}, S, \boldsymbol{s_1}, T_k, S_{bad} \rangle$ constructed as described below has a solution for some $k < \infty$.*

$$\begin{aligned} S &\coloneqq S^{(1)} \times \cdots \times S^{(m)} \\ \boldsymbol{s_1} &\coloneqq \langle \boldsymbol{s_1^{(1)}}, \ldots, \boldsymbol{s_1^{(m)}} \rangle \\ T_k(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y}) &\coloneqq \langle {T_k}^{(1)}(\boldsymbol{s^{(1)}}, \boldsymbol{x}, \boldsymbol{y}), \ldots, {T_k}^{(m)}(\boldsymbol{s^{(m)}}, \boldsymbol{x}, \boldsymbol{y}) \rangle \\ S_{bad} &\coloneqq \{ \boldsymbol{s} \mid \mathrm{idx}(\boldsymbol{s}) = k \} \end{aligned}$$

3.3 Extraction of Solutions

So far the results in Theorems 3 and 4 relate LTL realizability with the existence of solutions to the reduced safety games, but do not state how we can synthesize solutions to the original LTL specification. In effect, a winning strategy $f : (2^{\mathcal{X}})^+ \to 2^{\mathcal{Y}}$ that realizes the LTL specification $\langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ can be directly constructed from a policy $\pi : S \times X \to Y$ that solves the game G_k. Recursively,

$$\begin{aligned} f(\boldsymbol{x}) &\coloneqq \pi(\boldsymbol{s_1}, \boldsymbol{x}) \\ f(\boldsymbol{x_1} \cdots \boldsymbol{x_m}) &\coloneqq \pi(\boldsymbol{s_m}, \boldsymbol{x_m}) \\ \boldsymbol{s_{m+1}} &\coloneqq T(\boldsymbol{s_m}, \boldsymbol{x_m}, \pi(\boldsymbol{s_m}, \boldsymbol{x_m})) \end{aligned}$$

An advantageous property of solutions to the reduced safety games is that they provide means to implement f compactly in the form of a controller, or finite-state machine with internal memory that stores the current game state, and whose output to a sequence $\boldsymbol{x_1} \cdots \boldsymbol{x_m}$ depends only on the last environment move, $\boldsymbol{x_m}$, and the internal state $\boldsymbol{s_m}$ – not on the entire history.

3.4 Brief discussion on complexity

The transformation of LTL into an UCW automaton is worst-case exponential in the size of the formula. For a fixed $k < \infty$, the size of the state space is $\mathcal{O}(k^{\Sigma |Q_i|})$, that is, exponential in the sum of automata sizes. The game G_k can be solved by fix-point computation in polynomial time in the size of the search space. Decompositions of the LTL formula have the potential to produce smaller UCW automata and games with smaller size. Note, however, that the worst-case computational complexity is doubly exponential in the size of the LTL specification formula.

4 Dynamic Programming for LTL Synthesis

In Section 3 we showed how LTL synthesis can be cast as a series of safety games. Modern approaches to bounded synthesis use different technology (e.g. BDDs, SAT) to solve those games via logical inference (e.g. (Jobstmann and Bloem, 2006; Bohy et al., 2012)). Unfortunately, exact methods have limited scalability, because the size of the search space grows (worst-case) doubly-exponentially with the size of the LTL specification formula. In this section we study the use of dynamic programming to solve these safety games.

Dynamic programming gradually approximates optimal solutions, and is an alternative to exact methods that do logical inference. Dynamic programming has been widely used in the context of *Markov Decision Processes* (MDPs), where the objective is to compute strategies that optimize for reward-worthy behavior. The challenge for us in this work is to frame LTL synthesis as an optimization problem, for which dynamic programming techniques can be used. There are at least two significant differences between the dynamics of MDPs and safety games, that prevent us from using off-the-shelf methods. First, state transitions in an MDP are stochastic, whereas in a game the transitions are non-deterministic. Second, in an MDP the agent receives a reward signal upon performing an action, and optimizes for maximizing the expected (discounted) cumulative reward. In contrast, in a safety game the agent does not receive reward signals, and aims at winning the game. The same differences appear with so-called *Markov games* (Littman, 1994).

As it is common in dynamic programming, we first formulate a set of Bellman equations for safety games. Bellman equations describe the *value* of a state $\boldsymbol{s}$, $V(\boldsymbol{s})$, in terms of the values of future states in a sequential decision-making process, assuming both players act "optimally" – recall, the environment aims at maximizing the maximum index of visited states, and the system aims at keeping this number bounded. Solutions to the safety game can be obtained by performing, in each state, a greedy action that minimizes the (in our case, worst-case) value of the next state. Bellman equations can be solved using dynamic programming – for instance, value iteration. We show that value iteration on this set of Bellman equations is guaranteed to converge to a solution to the associated safety game. In the next section, we show how learning methods – in particular, an adaptation of (deep) Q-learning – can be used to provide guidance.

4.1 Bellman Equations

In our set of Bellman equations, we conceptualize $V(\boldsymbol{s})$ as a function that tells the maximum index of the states visited from $\boldsymbol{s}$, assuming that both players act optimally (cf. Theorem 5). Solutions are value – or, equivalently, Q-value – functions that satisfy the Bellman equations in each $\boldsymbol{s} \in S$.

Theorem 5. *An* LTL *specification $\langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ is realizable iff there exists a value function V with $V(\boldsymbol{s_1}) < \infty$ that is solution to the Bellman equations*

$$V(\boldsymbol{s}) = \max_{\boldsymbol{x}} Q(\boldsymbol{s}, \boldsymbol{x})$$
$$Q(\boldsymbol{s}, \boldsymbol{x}) = \max(\mathrm{idx}(\boldsymbol{s}), \min_{\boldsymbol{y}} V(T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y})))$$

Furthermore, policy $\pi(\boldsymbol{s}, \boldsymbol{x}) \coloneqq \arg\min_{\boldsymbol{y}} V(T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y}))$ solves the safety games G_k for $k \geq V(\boldsymbol{s_1})$.

Proof. If $\langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ is realizable, then the safety game G_k has a solution π for some $k < \infty$ (Theorem 3). $V(\boldsymbol{s})$ can be defined as the maximum index of the states reachable from $\boldsymbol{s}$ following π. In the other direction, let V be a solution to the Bellman equations. Then, the greedy policy $\pi(\boldsymbol{s}, \boldsymbol{x}) = \arg\min_y V(\delta(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y}))$ solves G_k for $k = V(\boldsymbol{s_1})$. □

Solutions to the safety games and, by extension, to the LTL synthesis problem can be constructed from solutions to the Bellman equations. In this step, it is crucial that the transition model (T) is known to the agent.

4.2 Value Iteration for Safety Games

Value iteration for MDPs updates the Q-values of states by performing one-step lookaheads. We naturally adapt this idea to develop a value iteration algorithm for safety games. The Q-value estimate for a pair $(\boldsymbol{s}, \boldsymbol{x})$ is $\hat{Q}(\boldsymbol{s}, \boldsymbol{x}) = \max(\mathrm{idx}(\boldsymbol{s}), \min_{\boldsymbol{y}} \max_{\boldsymbol{x'}} Q(T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y}), \boldsymbol{x'})$. Intuitively, $\hat{Q}(\boldsymbol{s}, \boldsymbol{x})$ performs one-step lookaheads from $\boldsymbol{s}$, and selects the lowest Q-value that the system player could enforce in an adversarial setting. Then, the Q-value $Q(\boldsymbol{s}, \boldsymbol{x})$ is updated to $\hat{Q}(\boldsymbol{s}, \boldsymbol{x})$. When the Bellman equations have a solution, the Q value updates converge in safe states and yield solutions to the associated safety games (Theorem 6).

Theorem 6. *If the Bellman equations in Theorem 5 have a solution, then the Q-value updates below make $V(\boldsymbol{s_1})$ converge to a bounded value, provided that the Q-value in each state is updated sufficiently often, and that the Q-value function is initialized to non-infinite values.*

$$Q(\boldsymbol{s}, \boldsymbol{x}) \leftarrow \max(\mathrm{idx}(\boldsymbol{s}), \min_{\boldsymbol{y}} \max_{\boldsymbol{x'}} Q(T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y}), \boldsymbol{x'})$$

Furthermore, policy $\pi(\boldsymbol{s}, \boldsymbol{x}) \coloneqq \arg\min_{\boldsymbol{y}} V(T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y}))$ converges to a solution the safety game G_k for some $k < \infty$.

Proof. We show that the Bellman backup operator $BV(\boldsymbol{s}) = \max_x \max(\mathrm{idx}(\boldsymbol{s}), \min_{\boldsymbol{y}} V(T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y}))$ is a *contraction* in the set of safe states $S_{safe} \subseteq S$. Let V^* be a solution to the Bellman equations, i.e., $BV^* = V^*$. By using the property $|\max_x f(x) - \max_x g(x)| \leq \max_x |f(x) - g(x)|$, and observing that V^* is finite in safe states, and infinite in unsafe states, it can be shown that $\max_{\boldsymbol{s} \in S_{safe}} |BV(x) - V^*(x)| \leq \max_{\boldsymbol{s} \in S_{safe}} |V(x) - V^*(x)|$. Hence, the Q value updates have to converge to finite values in the set of safe states, bounded by some $k < \infty$. It follows straightforward that the policy $\pi(\boldsymbol{s}, \boldsymbol{x}) \coloneqq \arg\min_{\boldsymbol{y}} V(T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y}))$ converges to a solution to the safety game G_k. □

5 DQS: Deep Q-learning for LTL Synthesis

State-of-the-art exact methods for LTL synthesis, and bounded synthesis in particular, have limited scalability. In part, this is due to the (potentially, doubly-exponential) size of the search space. We can expect, thus, similar challenges with the value iteration algorithm presented in Section 4. The objective in this paper is not to do value iteration for safety games. Rather, the Bellman equations (Theorems 5) and the convergence of value iteration (Theorem 6) set the theoretical foundations that we need to do (deep) Q-learning.

In this section we present a method to *learn* a Q function, with a variant of (deep) Q-learning adapted for safety games. Why do we want to do Q-learning, if we know the transition model? Our aim is not to learn a solution to the Bellman equations. Instead, we want to compute *good enough* approximations that provide good guidance. With this goal in mind, training a neural network seems to be a reasonable approach because neural networks can capture the structure of a problem and do inference. At inference time, the network provides guidance that can be used to construct solutions.

Our approach, which we describe below, constitutes the first attempt to integrate search and learning to do LTL synthesis. We provided and elegant and extensible algorithm with wich a variety of search and learning algorithms can be deployed. For example, the guidance obtained with the trained neural network can be used in combination with AND/OR search techniques such as AO*. Similarly, the guidance can be used as heuristics with more sophisticated search techniques such as LAO* (Hansen and Zilberstein, 2001) in service of LTL synthesis. In recent research we exploited the correspondence between LTL synthesis and AI automated planning (see, e.g., (Camacho and McIlraith, 2019)) to reduce LTL synthesis specifications into fully observable non-deterministic planning problems that can be solved with off-the-shelf planners (Camacho et al., 2018a,b). The heuristic obtained with our trained neural network can be used to guide planners, also in combination with domain-independent heuristics. Similar techniques than those that we present here can be used to learn heuristics and guide planners in planning problems with LTL goals (see, e.g., (Camacho et al., 2017; Camacho and McIlraith, 2019).

5.1 Deep Learning for LTL synthesis

We propose the use of a neural network to approximate the Q function. We mainly base our inspiration on recent success in *Deep Q-learning for MDPs* (DQN), where a neural network is used to approximate the Q function in reinforcement learning for MDPs (Mnih et al., 2015). Our approach shares other commonalities with existing differential approaches to program synthesis that use deep learning. Like the *Neural Turing Machine* (Graves et al., 2014), which augments neural networks with external memory. In comparison, our approach uses automata as compact memory.

For its similarities with DQN, we name our *Deep Q-learning for* LTL *Synthesis* approach DQS. Like DQN, DQS uses a neural network to approximate the Q function. The network is trained in batches of *states* obtained from a prioritized experience replay buffer. Experience is obtained by running search episodes, in which the moves of the environment and system players are simulated according to some exploration policy. The Q-values estimated by the neural network are used to guide search, as well as to extract solutions. Like DQN, DQS is not guaranteed to converge to a solution to the Bellman equations. However, *good enough* approximations may provide effective guidance, and yield correct solutions to the safety game and synthesis problems. We provide further details of the components of DQS below.

Q-value network: The Q-value network $\boldsymbol{Q}_\theta$, with parameters θ, takes as input a state vector $\boldsymbol{s}$, and outputs a vector $\boldsymbol{Q}_\theta(\boldsymbol{s}) = \langle Q_\theta(\boldsymbol{s}, 1), \ldots, Q_\theta(\boldsymbol{s}, D)\rangle$. Each $Q_\theta(\boldsymbol{s}, d)$ is an estimation of $Q(\boldsymbol{s}, \boldsymbol{x})$, if $\boldsymbol{x}$ is a binary representation of d. We sometimes abuse notation, and confuse d with its binary representation $\boldsymbol{x}$. The input of $\boldsymbol{Q}_\theta$ has the same dimension as the game states, i.e., the number of automaton states in $\mathcal{A}_\varphi$ – or the sum of automata states, if formula decompositions are exploited. The output has $D = 2^{|\mathcal{X}|}$ neurons. The network weights are intitialized to gaussian values close to zero.

Training episodes: Training episodes start in the initial state of the game, $\boldsymbol{s_1}$. Environment and system' actions are simulated according to an exploration policy. New states are generated according to the game transition function, T, and added into a prioritized experience replay buffer. Episodes last until a horizon bound is reached, or a state $\boldsymbol{s} = \langle -1, \ldots, -1\rangle$ is reached, or a state $\boldsymbol{s}$ with

$\mathrm{idx}(\boldsymbol{s}) > K$ is reached for some hyperparameter K. After that happens, a new training episode starts.

Exploration policy: Different exploration policies can be designed. Here, we use an adaptation of the epsilon-greedy policy commonly used in reinforcement learning. At the beginning of each episode, with probability μ, we set an *epsilon-greedy* exploration policy for the environment player. Otherwise, we set it to be *greedy*. We do the same for the system player, with independent probability. Greedy environment policies are $\pi(\boldsymbol{s}) = \arg\max_{\boldsymbol{x}} Q_\theta(\boldsymbol{s}, \boldsymbol{x})$. For the system, greedy policies are $\pi(\boldsymbol{s}, \boldsymbol{x}) = \arg\min_{\boldsymbol{y}} \max_{\boldsymbol{x'}} Q_\theta^{(\boldsymbol{x'})}(T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y}), \boldsymbol{x'})$. Epsilon-greedy policies select an action at random with probability ϵ, and otherwise act greedily. We use $\mu = \epsilon = 0.2$.

Batch learning step: Learning is done in batches of states, sampled from a prioritized experience replay buffer. A learning step involves computing, for each state $\boldsymbol{s}$ in the batch, a new estimate of the Q-values, $\hat{Q}_\theta(\boldsymbol{s}, \boldsymbol{x})$. The updates in each $\hat{Q}_\theta(\boldsymbol{s}, \boldsymbol{x})$ do a one-step lookahead as follows:

$$\begin{aligned}
\boldsymbol{Q}_\theta(\boldsymbol{s}) &\overset{\alpha}{\leftarrow} \langle \hat{Q}_\theta(\boldsymbol{s}, 1), \ldots, \hat{Q}_\theta(\boldsymbol{s}, 2^{|\mathcal{X}|}) \rangle \\
\hat{Q}_\theta(\boldsymbol{s}, \boldsymbol{x}) &= -1, \text{ if } \boldsymbol{s} = \langle -1, \ldots, -1 \rangle \\
\hat{Q}_\theta(\boldsymbol{s}, \boldsymbol{x}) &= \min(K, \max(\mathrm{idx}(\boldsymbol{s}), \\
&\qquad \min_{\boldsymbol{s'} = T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y})} \max_{\boldsymbol{x'}} \mathrm{round}(Q_\theta(\boldsymbol{s'}, \boldsymbol{x'})))), \text{ otherwise.}
\end{aligned}$$

where $\mathrm{round}(x)$ returns the closest integer to x. The values are rounded, as co-Büchi indexes are integers. To prevent the Q-value to be learned from diverging to infinite, we bound the values by $K < \infty$. The idea is to deem all states $\boldsymbol{s}$ with $V(\boldsymbol{s}) \geq K$ as losing states, and focus on producing good Q-value estimates in game states that have co-Büchi index lower than K. The batch learning step updates the network weigths as to minimize the sum of TD errors accross all the states in the batch, with some *learning rate* $\alpha \in (0, 1)$. The TD error in state $\boldsymbol{s}$ is $\Sigma_{\boldsymbol{x}} |Q(\boldsymbol{s}, \boldsymbol{x}) - \hat{Q}(\boldsymbol{s}, \boldsymbol{x})|$. To incentive solutions with low co-Büchi indexes, we perform L2 regularisation. We do a batch learning step every four exploration steps, which has reported good results in DQN (Mnih et al., 2015).

Prioritized experience replay buffer: Explored states $\boldsymbol{s}$ along episodes are added into a *prioritized experience replay* buffer (Schaul et al., 2016), with a preference value that equals their current TD error $\Sigma_{\boldsymbol{x}} |Q(\boldsymbol{s}, \boldsymbol{x}) - \hat{Q}(\boldsymbol{s}, \boldsymbol{x})|$. In a learning step, a batch of states is sampled in a manner that higher preference is given to those states with higher TD error. After a batch learning step, the preference value of the sampled states are updated to their new TD error.

Double DQS: We investigate an enhancement of DQS. We borrow ideas from *Double DQN* (DDQN) for MDPs, where a target network is used to stabilize learning (van Hasselt, 2010). In *Double DQS* (DDQS) we use a target network $\boldsymbol{Q}_t$ to estimate the Q-values in the one-step lookaheads, and update $\boldsymbol{Q}_t$ to $\boldsymbol{Q}_\theta$ at the end of each episode. In DDQS the equations of the batch learning step become:

$$\begin{aligned}
\hat{Q}_\theta(\boldsymbol{s}, \boldsymbol{x}) &= \min(K, \max(\mathrm{idx}(\boldsymbol{s}), \\
&\qquad \min_{\boldsymbol{s'} = T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y})} \max_{\boldsymbol{x'}} \mathrm{round}(Q_t(\boldsymbol{s'}, \boldsymbol{x'}))))
\end{aligned}$$

5.2 Solution extraction and verification

An advantage of knowing the transition function T is that solutions extracted from the neural network can be verified for correctness. At the end of each episode, we verify whether the greedy policy for the system player obtained from the Q-value network $\boldsymbol{Q}_\theta$ is a solution to the safety game G_K. Recall that a greedy sytem policy is $\pi(\boldsymbol{s}, \boldsymbol{x}) = \arg\min_{\boldsymbol{y}} \max_{\boldsymbol{x'}} Q_\theta^{(\boldsymbol{x'})}(T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y}), \boldsymbol{x'})$. The verification step can be performed by doing an exhaustive enumeration of all reachable game states $\boldsymbol{s}$ by π, from the initial state $\boldsymbol{s_1}$, and checking that all have $\mathrm{idx}(\boldsymbol{s}) < K$.

Learning from losing gameplays: The verification step fails when it encounters a losing partial play, that is, a partial play $\rho = \boldsymbol{s_1} \cdots \boldsymbol{s_n}$ with $\mathrm{idx}(\boldsymbol{s_n}) = K$. When this occurs, we run a series of n batch learning steps. Each batch learning step includes samples from the prioritized experience replay buffer and a state in ρ, starting backwards from $\boldsymbol{s_n}$.

5.3 Stronger Supervision Signals

One of the challenges in reinforcement learning is having to deal with sparse rewards. In our learning framework for safety games there are no rewards, but rather supervision signals that enforce the Q-value estimates in a state $\boldsymbol{s}$ are not lower than its co-Büchi index, $\mathrm{idx}(\boldsymbol{s})$. Arguably, the problem of sparsity may also manifest in our approach when the co-Büchi index of visited states along training episodes face sparse *phase transitions* (i.e., infrequent changes). In consequence, it may take a large number of episodes for these values to be propagated in the Q-value network.

We propose the use of a *potential* function $\Phi : S \to [0, 1)$ to provide stronger supervision signal in the learning process. Intuitively, $\Phi(\boldsymbol{s})$ indicates how close the co-Büchi index of a state $\boldsymbol{s} = \langle s^{(1)}, \ldots, s^{(m)} \rangle$ is from experiencing a phase transition that increments its value. Formally:

$$\Phi(\boldsymbol{s}) := \begin{cases} 0 & \text{if } \mathrm{idx}(\boldsymbol{s}) = -1 \\ \max\{1/(d(q_i)+1) | s^{(i)} = \mathrm{idx}(\boldsymbol{s})\} & \text{otherwise} \end{cases}$$

where $d(q)$ is the distance (minimum number of outer transitions) between automaton state q and a rejecting state. Without loss of generality, we presume in a UCW rejecting states are reachable from any state, and thus $d(q)$ is well defined.

The learning updates are redefined, as shown below, to include the supervision given by the potentials. Note, this time we take the integer part of Q_θ, $\mathrm{floor}(Q_\theta(\boldsymbol{s'}, \boldsymbol{x'}))$.

$$\begin{aligned} \boldsymbol{Q}_\theta(\boldsymbol{s}) &\xleftarrow{\alpha} \langle \hat{Q}_\theta(\boldsymbol{s}, 1), \ldots, \hat{Q}_\theta(\boldsymbol{s}, 2^{|\mathcal{X}|}) \rangle \\ \hat{Q}_\theta(\boldsymbol{s}, \boldsymbol{x}) &= -1, \text{ if } \boldsymbol{s} = \langle -1, \ldots, -1 \rangle \\ \hat{Q}_\theta(\boldsymbol{s}, \boldsymbol{x}) &= \min(K, \max(\mathrm{idx}(\boldsymbol{s}), \\ \min_{\boldsymbol{s'} := T(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{y})} & \max_{\boldsymbol{x'}}(\Phi(\boldsymbol{s'}) + \mathrm{floor}(Q_\theta(\boldsymbol{s'}, \boldsymbol{x'})))), \text{ otherwise.} \end{aligned}$$

6 Experiments

We implemented our Deep Q-learning approach to LTL synthesis. Code will be made publicly available, should the paper be accepted for publication. We used *Spot* to transform LTL formulae into UCW automata, and learning library Tensorflow 2.0. Experiments were conducted in 2.4GHz CPU Linux machines with 10GB of memory.

The purpose of our experiments was not to compete with state-of-the-art tools for LTL synthesis – these are much faster. Rather, we wanted to evaluate the potential for providing effective search guidance of our neural-based approach.

Hyperparameters: We experienced with different network sizes. Interestingly, we were able to learn good guidance with very small networks. We fixed a network with two dense hidden layers with as many neurons as input size – i.e., number of UCW automaton states. We used an ϵ-greedy exploration policy with $\mu = \epsilon = 0.2$, and Adam optimizer with learning rate adjusted to $\alpha = 0.005$. We set $K = 4$ and a horizon bound of 50 timesteps. These numbers were set large enough to find solutions. Search stopped after 1000 episodes. We used a prioritized experience replay buffer with batch size of 32 states. The learning step in $\boldsymbol{Q}_\theta$ was done every 4 timesteps. In DDQS, the target network $\boldsymbol{Q}_t$ was updated at the end of each episode.

Benchmarks: We evaluated our system on a family of 19 *lilydemo* benchmark problems retrieved from the LTL synthesis competition SYNTCOMP (SYNTCOMP, 2019).

Configurations: We tested the following configurations:

- DQS[$-$]: DQS, reusing losing gameplays for learning.
- DDQS: Implementation of DQS with a target network.
- DDQS[$-$]: DDQS, reusing losing gameplays for learning.
- DDQS[$-, \phi$]: Like DDQS[$-$], also using potentials.

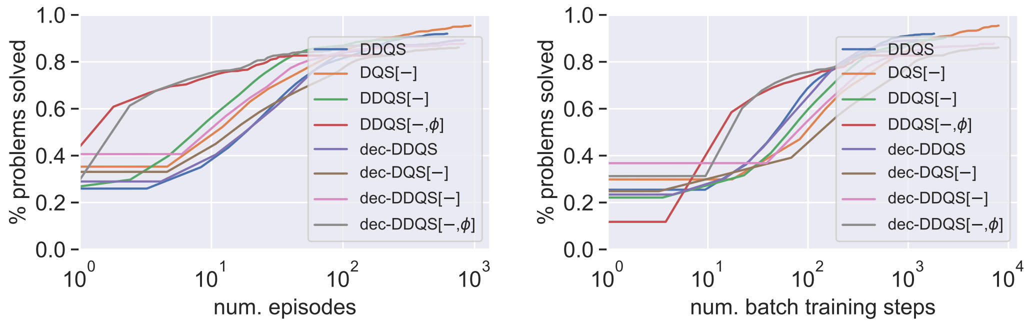

Figure 1: Problems solved wrt the number of episodes (left) and the number of batch learning steps (right).

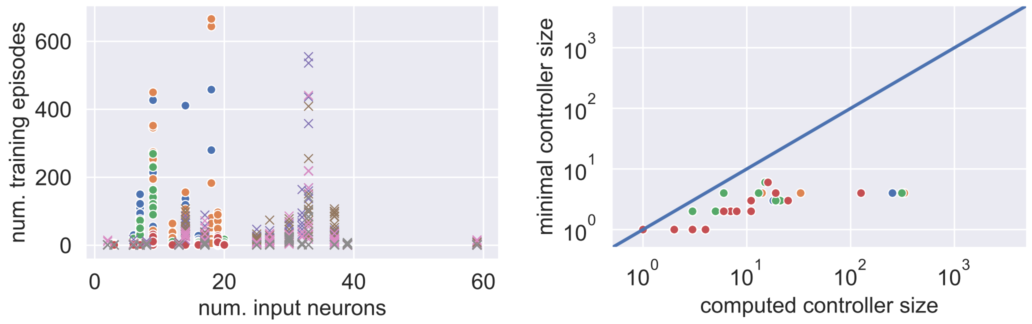

Figure 2: Left: training episodes wrt the input size of the neural network. Datapoints from algorithms that do not use decomposition are represented with circles, and otherwise represented with crosses. Right: controller sizes.

In addition, we tested the configurations above using automata decompositions of the LTL specification. We refer to those configurations as *dec-DDQS*, etc. Each configuration was tested in each benchmark a total of 20 times.

Impact of using a target network: The Q-values learned using a target network were more realistic than those learned without a target network, which tended to learn higher Q-values. These observed results resonate with the *optimistic* behaviour often manifested by DQN without the use of a target network. In terms of performance, DDQS was able to learn faster than DQS (i.e., wth a fewer number of episodes and batch training steps), but the differences were not huge in the benchmarks being tested (see Figure 1 (left and right)).

Impact of using decompositions: The use of automata decompositions may be necessary to scale to those problems with large specifications that, for computational limitations, cannot be transformed into a single automaton. Our learned greedy policies were not able to find exact solutions in large specifications where automata decompositions become necessary. Thus, further work needs be done to better exploit the guidance of the neural network in the search for solutions. Still, we conducted a study of the the impact of automata decompositions in the *lilydemo* benchmark set. The use of automata decompositions translated into a larger number of automaton states, and therefore, required a larger number of input neurons (see Figure 2 (left)). The use of automata decompositions also required more training episodes and steps, but not a huge number compared to single-automaton transformations of the LTL formula (see Figures 1 and 2).

Impact of using losing gameplays: By exploiting losing gameplays, our approaches were able to learn with fewer learning episodes and batch learning steps (see Figure 1).

Impact of using potentials: The use of a stronger supervision signal in the form of potentials greatly improved the learning process. We can observe in Figure 1 (left and right) that DDQS[$-$,ϕ] and dec-DDQS[$-$,ϕ] greatly outperformed all other configurations that do not make use of potentials.

Size of controllers: The controllers that we produced are significantly larger than the size of the minimal controllers for each benchmark problem (see Figure 2 (right)). This suggests that the performance of our system may benefit from the combination of more sophisticated search techniques – other than just epsilon-greedy exploration policies, and greedy execution policies – to prune the search space.

7 Discussion and Future Work

We addressed LTL synthesis, a 2EXP-complete type of program synthesis from specification that automatically generates programs that are correct by construction. Exact methods have limited scalability. The development of novel techniques with the potential to scale is crucial.

We presented the first approach to LTL synthesis that combines two scalable methods: search and learning. Our novel approach reformulates LTL synthesis as an optimization problem. We set the theoretical foundations to solve synthesis via dynamic programming, and explored deep Q-learning to approximate solutions. Ultimately, our objective was to train neural networks to provide good guidance. Our approach shares commonalities with neuro-symbolic and neural-guided search approaches in using learned properties to guide search. Like the *Neural Turing Machine*, which augments neural networks with external memory (Graves et al., 2014), we use *automata* as compact memory.

We were interested in evaluating the potential for providing effective search guidance of our neural-based approach. In our experiments, we solved synthesis benchmarks by virtue of simply executing policies that acted greedily wrt the neural network guidance. In many cases, the network was trained using only a few dozen episodes in problems whose solutions are simple, but the where size of the search space is $\mathcal{O}(2^{30})$ or more. Furthermore, we found that simple enhancements – like reusing losing plays – could significantly improve performance, and the use of potentials for stronger supervision signal proved to be extremely beneficial. While we did not solve the largest benchmark problems in SYNTCOMP, and our approach did not manifest state-of-the-art performance, we believe that the combination of learning and search proposed here provides a foundation for LTL synthesis and opens a new avenue for research.

References

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. (2017). Deepcoder: Learning to write programs. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*.

Bohy, A., Bruyère, V., Filiot, E., Jin, N., and Raskin, J. (2012). Acacia+, a tool for LTL synthesis. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, pages 652–657.

Camacho, A., Baier, J. A., Muise, C. J., and McIlraith, S. A. (2018a). Finite LTL synthesis as planning. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 29–38.

Camacho, A. and McIlraith, S. A. (2019). Strong fully observable non-deterministic planning with ltl and ltl-f goals. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 5523–5531.

Camacho, A., Muise, C. J., Baier, J. A., and McIlraith, S. A. (2018b). LTL realizability via safety and reachability games. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4683–4691.

Camacho, A., Triantafillou, E., Muise, C. J., Baier, J. A., and McIlraith, S. A. (2017). Non-deterministic planning with temporally extended goals: LTL over finite and infinite traces. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI)*, pages 3716–3724.

Chen, X., Liu, C., and Song, D. (2018). Towards synthesizing complex programs from input-output examples. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*.

Church, A. (1957). Applications of recursive arithmetic to the problem of circuit synthesis. *Summaries of the Summer Institute of Symbolic Logic, Cornell University 1957*, 1:3–50.

Ellis, K., Solar-Lezama, A., and Tenenbaum, J. (2016). Sampling for bayesian program learning. In *Proceedings of the 28th Annual Conference on Advances in Neural Information Processing Systems (NIPS)*, pages 1289–1297.

Gaunt, A. L., Brockschmidt, M., Singh, R., Kushman, N., Kohli, P., Taylor, J., and Tarlow, D. (2016). Terpret: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428.

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. *CoRR*, abs/1410.5401.

Gulwani, S. (2016). Programming by examples - and its applications in data wrangling. In *Dependable Software Systems Engineering*, pages 137–158.

Hansen, E. A. and Zilberstein, S. (2001). Lao*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62.

Jacobs, S., Bloem, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, P. J., Michaud, T., Sakr, M., Sickert, S., Tentrup, L., and Walker, A. (2019). The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results. *CoRR*, abs/1904.07736.

Jobstmann, B. and Bloem, R. (2006). Optimizations for LTL synthesis. In *Proceedings of the 6th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 117–124.

Kupferman, O. and Vardi, M. Y. (2005). Safraless decision procedures. In *Proceedings of the 46th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 531–542.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the 11th International Conference on Machine Learning (ICML)*, pages 157–163.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Parisotto, E., Mohamed, A., Singh, R., Li, L., Zhou, D., and Kohli, P. (2017). Neuro-symbolic program synthesis. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*.

Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 46–57.

Pnueli, A. and Rosner, R. (1989). On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–190.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized experience replay. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*.

Schewe, S. and Finkbeiner, B. (2007). Bounded synthesis. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 474–488.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362:1140–1144.

SYNTCOMP (2019). The reactive synthesis competition. `http://www.syntcomp.org`. Accessed: 2019-10-31.

van Hasselt, H. (2010). Double q-learning. In *Proceedings of the 24th Annual Conference on Advances in Neural Information Processing Systems (NIPS)*, pages 2613–2621.

Zhang, L., Rosenblatt, G., Fetaya, E., Liao, R., Byrd, W. E., Might, M., Urtasun, R., and Zemel, R. S. (2018). Neural guided constraint logic programming for program synthesis. In *Proceedings of the 31st Annual Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 1744–1753.